

# APÉNDICE 2

## ESPECIFICACIONES Y API

### 1.1. Lección 4

#### 1.1.1. API de tratamiento de cadenas de caracteres

Las siguientes funciones deben programarse en el archivo `strings.hpp`.

##### 1.1.1.1. Función `length`

*Prototipo:* `int length(string s);`

*Descripción:* Cuenta la cantidad de caracteres que componen la cadena `s`.

*Parámetro:* `string s` – Cadena cuya longitud debemos averiguar.

*Retorna:* `int` – Cuántos caracteres contiene la cadena `s`.

Ejemplo de uso:

```
string s = "Hola";  
int n = length(s);  
cout << n << endl; // muestra: 4
```

```
s = "";
n = length(s);
cout << n << endl; // muestra: 0
```

### 1.1.1.2. Función charCount

*Prototipo:* `int charCount(string s, char c);`

*Descripción:* Cuenta la cantidad de veces que aparece el carácter `c` dentro de `s`.

*Parámetros:*

- `string s` – Cadena que contiene al carácter `c`.
- `char c` – Carácter cuya cantidad de ocurrencias queremos averiguar.

*Retorna:* `int` – Cuántas veces aparece `c` dentro de la cadena `s`.

Ejemplo de uso:

```
string s = "Esto es una prueba";
int n = charCount(s, 'e');

cout << n << endl; // muestra: 2

n = charCount(s, ' ');
cout << n << endl; // muestra: 3
```

### 1.1.1.3. Función substring

*Prototipo:* `string substring(string s, int d, int h);`

*Descripción:* Retorna la subcadena de `s` comprendida entre las posiciones `d` (inclusive) y `h` (no inclusive).

*Parámetros:*

- `string s` – Cadena que contiene la subcadena que queremos obtener.
- `int d` – Posición (inclusive) que indica dónde comienza la subcadena.
- `int h` – Posición (no inclusive) que indica dónde finaliza la subcadena.

*Retorna:* `string` – La subcadena de `s` comprendida comprendida entre las posiciones `d` (inclusive) y `h` (no inclusive).

Ejemplo de uso:

```
string s = "Esto es una prueba";
string x = substring(s,2,10);
cout << x << endl; // muestra: to es un

x = substring(s,2,length(s));
cout << x << endl; // muestra: to es una prueba
```

#### 1.1.1.4. Función substring (sobrecarga)

*Prototipo:* `string substring(string s,int d);`

*Descripción:* Retorna la subcadena de `s` comprendida entre la posición `d` y el final de la cadena.

*Parámetros:*

- `string s` – Cadena que contiene la subcadena que queremos obtener.
- `int d` – Posición (inclusive) que indica dónde comienza la subcadena.

*Retorna:* `string` – La subcadena de `s` comprendida comprendida entre la posición `d` (inclusive) y el final de la cadena.

Ejemplo de uso:

```
string s = "Esto es una prueba";
string x = substring(s,2);
cout << x << endl; // muestra: to es una prueba
```

#### 1.1.1.5. Función indexOf

*Prototipo:* `int indexOf(string s,char c);`

*Descripción:* Retorna la posición que ocupa la primera ocurrencia del carácter `c` dentro de la cadena `s`.

*Parámetros:*

- `string s` – Cadena que contiene al carácter `c`.
- `char c` – Carácter cuya posición, dentro de `s`, queremos averiguar.

*Retorna:* `int` - La posición que ocupa, dentro de `s`, la primera ocurrencia del carácter `c`, o un valor negativo si `s` no contiene a `c`.

Ejemplo de uso:

```
string s = "Esto es una prueba";
int p = indexOf(s, 'e');
cout << p << endl; // muestra: 5

p = indexOf(s, 'X');
cout << p << endl; // muestra: -1
```

#### 1.1.1.6. Función `indexOf` (sobrecarga)

*Prototipo:* `int indexOf(string s, char c, int offset);`

*Descripción:* Retorna la posición que ocupa la primera ocurrencia de un carácter `c` dentro de la cadena `s`, descartando los primeros `offset` caracteres (desplazamiento inicial).

*Parámetros:*

- `string s` - Cadena que contiene al carácter `c`.
- `char c` - Carácter cuya posición, dentro de `s`, queremos averiguar.
- `int offset` - Posición de desplazamiento desde donde debemos buscar.

*Retorna:* `int` - La posición que ocupa, dentro de `s`, la primera ocurrencia del carácter `c`, considerando desde `offset`.

Ejemplo de uso:

```
string s = "Esto es una prueba";
int p = indexOf(s, 'e', 0);
cout << p << endl; // muestra: 5

p = indexOf(s, 'e', 12);
cout << p << endl; // muestra: 15
```

#### 1.1.1.7. Función `indexOf` (sobrecarga)

*Prototipo:* `int indexOf(string s, string toSearch);`

*Descripción:* Retorna la posición que ocupa la primera ocurrencia de `toSearch` dentro de la cadena `s`.

*Parámetros:*

- `string s` - Cadena que contiene al carácter `c`.
- `string toSearch` - Cadena cuya posición queremos averiguar.

*Retorna:* `int` - La posición inicial de la primera ocurrencia de `toSearch` dentro de `s` o un valor negativo si `s` no contiene a `toSearch`.

Ejemplo de uso:

```
string s = "Esto es una prueba";
int p = indexOf(s, "una");
cout << p << endl; // muestra: 8

p = indexOf(s, "jamon");
cout << p << endl; // muestra: algun valor negativo
```

#### 1.1.1.8. Función `indexOf` (sobrecarga)

*Prototipo:* `int indexOf(string s, string toSearch, int offset);`

*Descripción:* Retorna la posición que ocupa la primera ocurrencia de `toSearch` dentro de la cadena `s`, descartando los primeros `offset` caracteres (desplazamiento inicial).

*Parámetros:*

- `string s` - Cadena que contiene a `toSearch`.
- `string toSearch` - Cadena que vamos a buscar.
- `int offset` - Posición de desplazamiento desde donde debemos buscar.

*Retorna:* `int` - La posición que ocupa la primera ocurrencia de `toSearch` considerando a `s` a partir de la posición `offset`.

Ejemplo de uso:

```
string s = "Esta funcion es la funcion mas dificil";
int p = indexOf(s, "funcion", 0);
```

```
cout << p << endl; // muestra: 5
p = indexOf(s,"funcion",13);
cout << p << endl; // muestra: 19
```

#### 1.1.1.9. Función lastIndexOf

**Prototipo:** `int lastIndexOf(string s, char c);`

**Descripción:** Retorna la posición de la última ocurrencia del carácter `c` dentro de `s`.

**Parámetros:**

- `string s` - Cadena que contiene al carácter `c`.
- `char c` - Carácter cuya última posición, dentro de `s`, queremos averiguar.

**Retorna:** `int` - La posición que ocupa, dentro de `s`, la última ocurrencia del carácter `c`, o un valor negativo si `s` no contiene a `c`.

#### 1.1.1.10. Función indexOfN

**Prototipo:** `int indexOfN(string s, char c, int n);`

**Descripción:** Retorna la posición de la  $n$ -ésima ocurrencia de `c` dentro de `s`. Si  $n$  es 0 (cero) retorna -1; si  $n$  es mayor que la cantidad de ocurrencias de `c` retorna la longitud de la cadena `s`.

**Parámetros:**

- `string s` - Cadena que contiene al carácter `c`.
- `char c` - Carácter cuya posición se debe determinar. Se asume que `s` contiene a `c`, al menos,  $n$  veces.
- `int n` - Número de ocurrencia de `c`, contando desde 1.

**Retorna:** `int` - La posición de la  $n$ -ésima ocurrencia de `c` dentro de `s`.

**Ejemplo de uso:**

```
string s = "John|Paul|George|Ringo";
int p = indexOfN(s,'|',1);
cout << p << endl; // muestra: 4
```

```
p = indexOfN(s, '|', 2);
cout << p << endl; // muestra: 9

p = indexOfN(s, '|', 3);
cout << p << endl; // muestra: 16
```

#### 1.1.1.11. Función charToInt

*Prototipo:* `int charToInt(char c);`

*Descripción:* Retorna el valor numérico que representa el carácter `c`.

*Parámetro:* `char c` – Carácter que podrá ser numérico o alfabético, siendo `'0' <= c <= '9'`, `'A' <= c <= 'Z'`, `'a' <= c <= 'z'`. Acepta indistintamente mayúsculas y minúsculas.

*Retorna:* `int` - El valor numérico del carácter `c`. Para letras se considera: `'A'=10`.

Ejemplo de uso:

```
char c = '2';
int n = charToInt(c); // retorna: 2
cout << n << endl;

c = 'D';
n = charToInt(c); // retorna: 13
cout << n << endl;
```

#### 1.1.1.12. Función intToChar

*Prototipo:* `char intToChar(int i);`

*Descripción:* Retorna el carácter que representa al valor de `i`, que debe estar comprendido entre 0 y 9, o en entre 65 y 90. Es la función inversa de `charToInt`.

*Parámetro:* `int i` - Valor numérico.

*Retorna:* `char` - El carácter que representa al valor de `i`, considerando que `10='A'`.

Ejemplo de uso:

```
int i = 2;
char c = intToChar(i);
cout << c << endl; // muestra: 2
```

```
i = 13;
c = intToChar(i);
cout << c << endl; // muestra: D
```

#### 1.1.1.13. Función getDigit

*Prototipo:* `int getDigit(int n, int i);`

*Descripción:* Retorna el *i*-ésimo dígito del valor de *n*.

*Parámetros:*

- `int n` - Número entero de 1 o más dígitos desde donde se quiere obtener el dígito que se ubica en la *i*-ésima posición.
- `int i` - Posición, contando desde 0 (cero) y de derecha a izquierda, del dígito de *n* que queremos obtener.

*Retorna:* `int` - El dígito que se ubica en la *i*-ésima posición de *n*.

Ejemplo de uso:

```
int n = 12345;
int i = 0;
int r = getDigit(n, i);

cout << r << endl; // muestra: 5

i = 1;
r = getDigit(n, i);
cout << r << endl; // muestra: 4
```

#### 1.1.1.14. Función digitCount

*Prototipo:* `int digitCount(int n);`

*Descripción:* Retorna la cantidad de dígitos que contiene el valor de *n*.

*Parámetro:* `int n` - Valor numérico cuya cantidad de dígitos queremos averiguar.

*Retorna:* `int` - La cantidad de dígitos que tiene el valor de *n*.

Ejemplo de uso:



```
int n = 12345;
int i = digitCount(n);
cout << i << endl; // muestra: 5
```

#### 1.1.1.15. Función `intToString`

*Prototipo:* `string intToString(int i);`

*Descripción:* Retorna una cadena de caracteres representando el valor `i`.

*Parámetro:* `int i` - Valor numérico entero que se va a representar como `string`.

*Retorna:* `string` - Cadena de caracteres que representando el valor de `i`.

Ejemplo de uso:

```
int i = 12345;
string s = intToString(i);
cout << s << endl; // muestra: 12345
```

#### 1.1.1.16. Función `stringToInt`

*Prototipo:* `int stringToInt(string s, int b);`

*Descripción:* Retorna el valor numérico representado en la cadena `s`, considerando que dicho valor está expresado en la base numérica `b`.

*Parámetros:*

- `string s` - Cadena que representa un valor numérico entero en base `b`.
- `int b` - Base numérica del valor que representado en la cadena `s`.

*Retorna:* `int` - El número numérico representado en la cadena `s`.

Ejemplo de uso:

```
string s = "10";
int i = stringToInt(s, 10);
cout << i << endl; // muestra: 10

i = stringToInt(s, 2);
cout << i << endl; // muestra: 2
```

```
i = stringToInt(s,16);
cout << i << endl; // muestra: 16

s = "12AB";
i = stringToInt(s,16);
cout << i << endl; // muestra: 4779
```

#### 1.1.1.17. Función stringToInt (sobrecarga)

**Prototipo:** `int stringToInt(string s); // SOBRECARGA`

**Descripción:** Retorna el valor numérico de la cadena `s`, que sólo debe contener dígitos numéricos en base 10. Esta función es la función inversa de `intToString`.

**Parámetro:** `string s` - Cadena de caracteres que sólo contiene dígitos numéricos.

**Retorna:** `int` - El valor numérico que está representado en la cadena `s`.

Ejemplo de uso:

```
string s = "12345";

int i = stringToInt(s);
cout << i << endl; // muestra: 12345
```

#### 1.1.1.18. Función charToString

**Prototipo:** `string charToString(char c);`

**Descripción:** Retorna una cadena cuyo único carácter es `c`.

**Parámetro:** `char c` - Carácter que será el contenido de la cadena.

**Retorna:** `string` - Una cadena de longitud 1 cuyo único carácter será `c`.

Ejemplo de uso:

```
char c = 'A';
string s = charToString(c);
cout << s << endl; // muestra: A
cout << length(s) << endl; // muestra: 1
```

```

c = ' ';
string s = charToString(c);

cout << s << endl;           // muestra: [VACIO]
cout << length(s) << endl;   // muestra: 1

```

#### 1.1.1.19. Función stringToChar

**Prototipo:** `char stringToChar(string s);`

**Descripción:** Retorna el único carácter que contiene la cadena `s`. Esta es la función inversa de `charToString`.

**Parámetro:** `string s` - Cadena de caracteres de longitud 1.

**Retorna:** `char` - El único carácter que contiene la cadena `s`.

Ejemplo de uso:

```

string s = "A";
char c = stringToChar(s);

cout << c << endl;           // muestra: A
cout << (int)c << endl;       // muestra: 65

s = " ";
c = stringToChar(s);
cout << c << endl;           // muestra: [VACIO]
cout << (int)c << endl;       // muestra: 32, ASCII de ' '

```

#### 1.1.1.20. Función stringToString

**Prototipo:** `string stringToString(string s);`

**Descripción:** Retorna la misma cadena que recibe. Se trata de una función trivial que usaremos más adelante, dentro de este mismo capítulo.

**Parámetro:** `string s` - Cadena de caracteres.

**Retorna:** `string` - La misma cadena que recibe como parámetro.

Ejemplo de uso:

```
string s = toString("Hola");
cout << s << endl; // muestra: Hola
```

#### 1.1.1.21. Función doubleToString

*Prototipo:* `string doubleToString(double d);`

*Descripción:* Retorna una cadena representando el valor contenido en `d`.

*Parámetro:* `double d` - Valor que se representará como cadena.

*Retorna:* `string` - Cadena de caracteres representando el valor de `d`.

Ejemplo de uso:

```
double d = 123.4;
string s = doubleToString(d);
cout << s << endl; // muestra: 123.4
```

#### 1.1.1.22. Función stringToDouble

*Prototipo:* `double stringToDouble(string s);`

*Descripción:* Retorna el valor numérico representado en la cadena `s`.

*Parámetro:* `string s` - Cadena que contiene un valor compatible con `double`.

*Retorna:* `double` - El valor que está representado en la cadena `s`.

Ejemplo de uso:

```
string s = "123.4";
double d = stringToDouble(s);
cout << d << endl; // muestra: 123.4
```

#### 1.1.1.23. Función isEmpty

*Prototipo:* `bool isEmpty(string s);`

*Descripción:* Retorna `true` o `false` según `s` sea o no la cadena vacía.

*Parámetro:* `string s` - Una cadena de caracteres.

*Retorna:* bool – Retorna true si s es la cadena vacía o false si no lo es.

Ejemplo de uso:

```
string s = "";
cout << isEmpty(s) << endl; // true

s = "Hola";
cout << isEmpty(s) << endl; // false

s = "    ";
cout << isEmpty(s) << endl; // false
```

#### 1.1.1.24. Función startsWith

*Prototipo:* bool startsWith(string s, string x);

*Descripción:* Determina si x es prefijo de s.

*Parámetros:*

- string s - Cadena que podría comenzar con x.
- char x - Cadena que podría ser prefijo de s.

*Retorna:* bool - true si x es prefijo de s.

Ejemplo de uso:

```
string s1 = "cursoDeAlgoritmos";
string s2 = "curso";

if( startsWith(s1,s2) )
{
    cout << s2 << " es prefijo de: " << s1 << endl;
}
```

#### 1.1.1.25. Función endsWith

*Prototipo:* bool endsWith(string s, string x);

*Descripción:* Determina si x es sufijo de s.

*Parámetros:*

- `string s` - Cadena que podría finalizar con `x`.
- `char x` - Cadena que podría ser sufijo de `s`.

*Retorna:* `bool` - `true` si `x` es sufijo de `s`.

Ejemplo de uso:

```
string s1 = "cursoDeAlgoritmos";
string s2 = "Algoritmos";

if( endsWith(s1,s2) )
{
    cout << s2 << " es sufijo de: " << s1 << endl;
}
```

#### 1.1.1.26. Función `contains`

*Prototipo:* `bool contains(string s, char c);`

*Descripción:* Determinar si la cadena `s` contiene al carácter `c`.

*Parámetros:*

- `string s` - Cadena que podría contener al carácter `c`.
- `char c` - Carácter cuyo valor podría estar contenido en `s`.

*Retorna:* `bool` - `true` si `s` contiene a `c`; `false` si no lo contiene.

Ejemplo de uso:

```
string s = "abcd";
char c = 'b';

if( contains(s,c) )
{
    cout << s << " contiene a: " << c << endl;
}

c = 'X';
if( !contains(s,c) )
{
    cout << s << " NO contiene a: " << c << endl;
}
```

**1.1.1.27. Función replace**

*Prototipo:* `string replace(string s, char oldChar, char newChar);`

*Descripción:* Reemplaza en `s` todas las ocurrencias de `oldChar` por `newChar`.

*Parámetros:*

- `string s` - Cadena sobre la cual se reemplazarán los caracteres.
- `char oldChar` - Carácter que va a ser reemplazado por `newChar`.
- `char newChar` - Valor que reemplazará todas las ocurrencias de `oldChar`.

*Retorna:* `string` - La cadena `s` con caracteres `oldChar` donde antes tenía `oldChar`.

*Ejemplo de uso:*

```
string s = "Esto es una prueba";
string r = replace(s, 'e', 'X');
cout << r << endl; // SALIDA: Esto Xs una pruxba
```

**1.1.1.28. Función insertAt**

*Prototipo:* `string insertAt(string s, int pos, char c);`

*Descripción:* Insertar el carácter `c` en la posición `pos` de la cadena `s`.

*Parámetros:*

- `string s` - Cadena de caracteres donde se insertará un carácter.
- `int pos` - Posición de `s` se va a insertar al carácter `c`.
- `char c` - Carácter que se insertará en `s`, en la posición `pos`.

*Retorna:* `string` - Una cadena cuya longitud será `length(s) + 1`, idéntica a `s` pero con el valor de `c` insertado en la posición `pos`.

*Ejemplo de uso:*

```
string s = "Esto es una prueba";
int pos = 6;

char c = 'X';
string r = insertAt(s, pos, c);
```

```
cout << r << endl; // SALIDA: Esto eXs una prueba
```

#### 1.1.1.29. Función removeAt

*Prototipo:* `string removeAt(string s, int pos);`

*Descripción:* Remover de `s` el carácter ubicado en la posición `pos`.

*Parámetros:*

- `string s` - Cadena de caracteres sobre la cual se removerá un carácter.
- `int pos` - Posición del carácter que se removerá.

*Retorna:* `string` - Una cadena igual a `s` pero sin `s[pos]`.

Ejemplo de uso:

```
string s = "Esto es una prueba";
int pos = 7;

string r = removeAt(s, pos);

cout << r << endl; // SALIDA: Esto esuna prueba
```

#### 1.1.1.30. Función ltrim

*Prototipo:* `string ltrim(string s);`

*Descripción:* Recorta los espacios en blanco que se encuentren a la izquierda de `s`.

*Parámetro:* `string s` - Cadena que podría tener espacios a la izquierda.

*Retorna:* `string` - Una cadena idéntica a `s` pero sin espacios a la izquierda.

Ejemplo de uso:

```
// con espacios a izquierda
string s = "    Esto es una prueba";
string r = ltrim(s);
cout << "[" << r << "]" << endl; // [Esto es una prueba]
```



```
// sin espacios
s = "Esto es una prueba";
r = ltrim(s);
cout << "[" << r << "]" << endl; // [Esto es una prueba]

// con espacios a izquierda y derecha
s = "    Esto es una prueba    ";
r = ltrim(s);
cout << "[" << r << "]" << endl; // [Esto es una prueba    ]
```

#### 1.1.1.31. Función rtrim

**Prototipo:** string rtrim(string s);

**Descripción:** Recortar los espacios en blanco a la derecha de s.

**Parámetro:** string s - Cadena que podría tener espacios a la derecha.

**Retorna:** string - Una cadena idéntica a s sin espacios en blanco a la derecha.

Ejemplo de uso:

```
// con espacios a derecha
string s = "Esto es una prueba    ";
string r = rtrim(s);
cout << "[" << r << "]" << endl; // [Esto es una prueba]

// sin espacios
s = "Esto es una prueba";
r = rtrim(s);
cout << "[" << r << "]" << endl; // [Esto es una prueba]

// con espacios a izquierda y derecha
s = "    Esto es una prueba    ";
r = rtrim(s);
cout << "[" << r << "]" << endl; // [    Esto es una prueba]
```

#### 1.1.1.32. Función trim

**Prototipo:** string trim(string s);

**Descripción:** Recortar los espacios en blanco ubicados a izquierda y derecha de s.

**Parámetro:** `string s` - Cadena que podría contener espacios en los extremos.

**Retorna:** `string` - Una cadena idéntica a `s` sin espacios en los extremos.

Ejemplo de uso:

```
// con espacios a izquierda y derecha
string s = "    Esto es una prueba    ";
string r = trim(s);
cout << "[" << r << "]" << endl; // [Esto es una prueba]

// con espacios dentro de la cadena
s = "Esto    es una prueba";
r = rtrim(s);
cout << "[" << r << "]" << endl; // [Esto    es una prueba]
```

#### 1.1.1.33. Función replicate

**Prototipo:** `string replicate(char c,int n);`

**Descripción:** Generar una cadena de caracteres compuesta por `n` caracteres `c`.

**Parámetros:**

- `char c` - Carácter que se replicará `n` veces para generar la cadena.
- `int n` - Cantidad de caracteres que tendrá la cadena generada.

**Retorna:** `string` - Una cadena compuesta por `n` caracteres `c`.

Ejemplo de uso:

```
int n = 5;
char c = 'X'
string r = replicate(c,n);

cout << "[" << r << "]" << endl; // muestra: [XXXXX]

c = ' ';
r = replicate(c,n);
cout << "[" << r << "]" << endl; // muestra: [      ]
```

#### 1.1.1.34. Función spaces

**Prototipo:** `string spaces(int n);`

*Descripción:* Genera una cadena de caracteres compuesta por  $n$  caracteres ' '.

*Parámetro:* `int n` - Longitud de la cadena que se generará.

*Retorna:* `string` - Una cadena compuesta por  $n$  caracteres ' '.

Ejemplo de uso:

```
int n = 5;
string r = spaces(n);

// muestra: [      ] (cinco espacios)
cout << "[" << r << "]" << endl;
```

#### 1.1.1.35. Función `lpad`

*Prototipo:* `string lpad(string s, int n, char c);`

*Descripción:* Retorna una cadena idéntica a `s`, con longitud `n` completando, si fuese necesario, con caracteres `c` a la izquierda hasta llegar a la longitud requerida.

*Parámetros:*

- `int n` - Longitud final que tendrá la cadena.
- `char c` - Carácter con que se completará a `s` si fuera necesario.

*Retorna:* `string` - Una cadena de longitud `n` compuesta por `n-length(s)` caracteres `c` seguidos de la cadena `s`.

Ejemplo de uso:

```
string s = "Hola";
int n = 10;
char c = 'X';

string r = lpad(s, n, c);

cout << "[" << r << "]" << endl; // muestra: [XXXXXXHola]
```

#### 1.1.1.36. Función `rpadd`

*Prototipo:* `string rpadd(string s, int n, char c);`

*Descripción:* Idem `lpad` pero, de ser necesario, agrega caracteres `c` a la derecha.

*Parámetros:*

- `int n` - Longitud final que tendrá la cadena retornada.
- `char c` - Carácter con que se debe completará la cadena.

*Retorna:* `string` - Una cadena de longitud `n` compuesta por el contenido de `s` seguida de `n-length(s)` caracteres `c`.

Ejemplo de uso:

```
string s = "Hola";
int n = 10;
char c = 'X';

string r = rpad(s,n,c);

cout << "[" << r << "]" << endl; // muestra: [HolaXXXXXX]
```

### 1.1.1.37. Función `cpad`

*Prototipo:* `string cpad(string s,int n,char c);`

*Descripción:* Idem `rpadd` pero distribuye los caracteres `c` a izquierda y derecha.

*Parámetros:*

- `int n` - Longitud final que tendrá la cadena.
- `char c` - Carácter con que se completará si fuere necesario.

*Retorna:* `string` - Una cadena de longitud `n` compuesta por `s` y caracteres `c` distribuidos a la izquierda y a la derecha de modo tal que su longitud final sea `n`.

Ejemplo de uso:

```
string s = "Hola";
int n = 10;
char c = 'X';

string r = cpad(s,n,c);
cout << "[" << r << "]" << endl; // muestra: [XXXHolaXXX]
```

**1.1.1.38. Función isDigit**

*Prototipo:* `bool isDigit(char c);`

*Descripción:* Determinar si el valor de `c` corresponde o no a un dígito numérico.

*Parámetro:* `char c` - Carácter a determinar si representa a un dígito numérico.

*Retorna:* `bool` - `true` si `c` es '0', '1', '2', ..., '9', `false` en cualquier otro caso.

Ejemplo de uso:

```
char c = '9';
if( isDigit(c) )
{
    cout << c << " es digito" << endl; // SALIDA
}

c = 'A';
if( !isDigit(c) )
{
    cout << c << " NO es digito" << endl; // SALIDA
}
```

**1.1.1.39. Función isLetter**

*Prototipo:* `bool isLetter(char c);`

*Descripción:* Determina si el valor de `c` corresponde o no a una letra.

*Parámetro:* `char c` - Carácter a determinar si representa a una letra.

*Retorna:* `bool` - `true` si `c` es 'A', 'B', 'C', ..., 'Z' o 'a', 'b', 'c', ..., 'z'. Si no retorna `false`.

Ejemplo de uso:

```
char c = 'X';
if( isLetter(c) )
{
    cout << c << " es letra" << endl; // SALIDA
}

c = '9';
if( !isLetter(c) )
{
```

```
cout << c << " NO es letra" << endl; // SALIDA
}
```

#### 1.1.1.40. Función isUpperCase

*Prototipo:* `bool isUpperCase(char c);`

*Descripción:* Determinar si el valor de `c` corresponde a una letra mayúscula.

*Parámetro:* `char c` - Carácter para determinar si es una letra mayúscula.

*Retorna:* `bool` - `true` si `c` es 'A', 'B', 'C', ..., 'Z', `false` en cualquier otro caso.

Ejemplo de uso:

```
char c = 'X';
if( isUpperCase(c) )
{
    cout << c << " es letra mayuscula" << endl; // SALIDA
}

c = 'x';
if( !isUpperCase(c) )
{
    cout << c << " NO es mayuscula" << endl; // SALIDA
}
```

#### 1.1.1.41. Función isLowerCase

*Prototipo:* `bool isLowerCase(char c);`

*Descripción:* Determina si el valor de `c` corresponde a una letra minúscula.

*Parámetro:* `char c` - Carácter a debe determinar si contiene una letra minúscula.

*Retorna:* `bool` - `true` si `c` es 'a', 'b', 'c', ..., 'z', `false` en cualquier otro caso.

Ejemplo de uso:

```
char c = 'a';
if( isLowerCase(c) )
{
```

```

    cout << c << " es letra minuscula" << endl; // SALIDA
}

c = 'A';
if( !isLowerCase(c) )
{
    cout << c << " NO es minuscula" << endl; // SALIDA
}

```

#### 1.1.1.42. Función toUpperCase

**Prototipo:** `char toUpperCase(char c);`

**Descripción:** Convertir el valor de `c` a mayúscula.

**Parámetro:** `char c` - El carácter cuyo valor se debe convertir a mayúscula.

**Retorna:** `char` - Si `c` es una letra minúscula retorna su mayúscula, en cualquier otro caso retorna el mismo valor de `c`.

Ejemplo de uso:

```

char c = 'a';
char r = toUpperCase(c);
cout << r << endl; // Salida: A (convierte a mayuscula)

c = 'B';
r = toUpperCase(c);
cout << r << endl; // Salida: B (ya era mayuscula)

c = '9';
r = toUpperCase(c);
cout << r << endl; // Salida: 9 (no es una letra)

```

#### 1.1.1.43. Función toLowerCase

**Prototipo:** `char toLowerCase(char c);`

**Descripción:** Convierte el valor de `c` a minúscula.

**Parámetro:** `char c` - El carácter cuyo valor se debe convertir a minúscula.

*Retorna:* char – Si *c* es una letra mayúscula retorna su minúscula, en cualquier otro caso retorna el mismo carácter *c*.

Ejemplo de uso:

```
char c = 'A';
char r = toLowerCase(c);
cout << r << endl; // Salida: a (convierte a minúscula)

c = 'b';

r = toLowerCase(c);
cout << r << endl; // Salida: b (ya era minúscula)

c = '9';
r = toLowerCase(c);
cout << r << endl; // Salida: 9 (no es una letra)
```

#### 1.1.1.44. Función toUpperCase (sobrecarga)

*Prototipo:* string toUpperCase(string s);

*Descripción:* Retorna una cadena idéntica a *s* pero completamente en mayúsculas.

*Parámetro:* string *s* – Cadena cuyo valor se debe convertir a mayúscula.

*Retorna:* string – Una cadena igual a *s* pero totalmente en mayúsculas.

Ejemplo de uso:

```
string s = "hola";
string r = toUpperCase(s);
cout << r << endl; // Salida: HOLA
```

#### 1.1.1.45. Función toLowerCase (sobrecarga)

*Prototipo:* string toLowerCase(string s);

*Descripción:* Retorna una cadena idéntica a *s* pero completamente en minúsculas.

*Parámetro:* string *s* – Cadena cuyo valor se debe convertir a minúsculas.

*Retorna:* string – Una cadena igual a *s* pero totalmente en minúsculas.



Ejemplo de uso:

```
string s = "HOLA";
string r = toLowerCase(s);
cout << r << endl; // Salida: hola
```

#### 1.1.1.46. Función cmpString

*Prototipo:* `int cmpString(string a, string b);`

*Descripción:* Compara alfabéticamente dos cadenas.

*Parámetros:*

- `string a` – Cadena a comparar.
- `string b` – Cadena a comparar.

*Retorna:* `int` – Un valor negativo si `a` es alfabéticamente menor que `b`. Un valor positivo si `a` es alfabéticamente mayor que `b`, o 0 si ambas cadenas son iguales.

Ejemplo de uso:

```
string s1 = "Carlos";
string s2 = "Pablo";

if( cmpString(s1,s2)<0 )
{
    cout << s1 << " es menor que: " << s2 << endl;
}
```

#### 1.1.1.47. Función cmpDouble

*Prototipo:* `int cmpDouble(double a, double b);`

*Descripción:* Compara dos valores.

*Parámetros:*

- `double a` – Valor a comparar.
- `double b` – Valor a comparar.

*Retorna:* `int` – Un valor negativo si `a` es menor que `b`. Un valor positivo si `a` es mayor que `b`, o 0 si ambos valores son iguales.

Ejemplo de uso:

```
double x = 25.7;
double y = 36.9;

if( cmpDouble(x,y)<0 )
{
    cout << x << " es menor que: " << y << endl;
}
```

## 1.2. Lección 5

### 1.2.1. API de tratamiento de tokens

Las siguientes funciones deben programarse en el archivo `tokens.hpp`.

#### 1.2.1.1. Función `tokenCount`

*Prototipo:* `int tokenCount(string s, char sep);`

*Descripción:* Cuenta la cantidad *tokens* que el separador *sep* genera en *s*.

*Parámetros:*

- `string s` – Cadena *tokenizada*.
- `char sep` – Carácter separador.

*Retorna:* `int` – Cuántos *tokens* genera *sep* en la cadena *s*.

Ejemplo de uso:

```
string s = "John|Paul|George|Ringo";
char sep = '|';
int n = tokenCount(s,sep);
cout << n << endl; // Salida: 4

s = "John";
sep = '|';
n = tokenCount(s,sep);
cout << n << endl; // Salida: 1
```

```
s = "";
sep = '|';
n = tokenCount(s, sep);

cout << n << endl; // Salida: 0
```

#### 1.2.1.2. Función addToken

**Prototipo:** void addToken(string& s, char sep, string t);

**Descripción:** Agrega el token *t* al final de la cadena *s*.

**Parámetros:**

- string& s - Cadena tokenizada.
- char sep - Carácter separador.
- string t - Token que se agregará al final de s.

**Retorna:** void.

**Ejemplo de uso:**

```
string s = "";
char sep = '|';

addToken(s, sep, "John");
cout << s << endl; // Salida: John

addToken(s, sep, "Paul");
cout << s << endl; // Salida: John|Paul

addToken(s, sep, "George");
cout << s << endl; // Salida: John|Paul|George

addToken(s, sep, "Ringo");
cout << s << endl; // Salida: John|Paul|George|Ringo
```

#### 1.2.1.3. Función getTokenAt

**Prototipo:** string getTokenAt(string s, char sep, int i);

**Descripción:** Retorna el *i*-ésimo token de la cadena tokenizada *s*.

Parámetros:

- `string s` - Cadena *tokenizada*.
- `char sep` - Carácter separador.
- `int i` - Posición del *token* que se quiere obtener comenzando desde la izquierda y contando a partir de 0 (cero).

*Retorna:* `string` - El token ubicado en la posición *i* de la cadena *s*.

Ejemplo de uso:

```
string s = "John|Paul|George|Ringo";
char sep = '|';
int pos = 0;

string t = getTokenAt(s, sep, pos);
cout << t << endl; // Salida: John

pos = 1;
t = getTokenAt(s, sep, pos);
cout << t << endl; // Salida: Paul

pos = 2;
t = getTokenAt(s, sep, pos);
cout << t << endl; // Salida: George

pos = 3;
t = getTokenAt(s, sep, pos);
cout << t << endl; // Salida: Ringo
```

#### 1.2.1.4. Función `removeTokenAt`

*Prototipo:* `void removeTokenAt(string& s, char sep, int i);`

*Descripción:* Remueve de *s* el *token* ubicado en la posición *i*.

Parámetros:

- `string& s` - Cadena *tokenizada*.
- `char sep` - Carácter separador.
- `char i` - Posición del *token* que será removido de la cadena *s*.

*Retorna:* `void`.

Ejemplo de uso:

```
string s = "John|Paul|George|Ringo";
char sep = '|';

int i = 2;
removeTokenAt(s, sep, i);
cout << s << endl; // Salida: John|Paul|Ringo

i = 0;
removeTokenAt(s, sep, i);
cout << s << endl; // Salida: Paul|Ringo
```

#### 1.2.1.5. Función setTokenAt

*Prototipo:* void setTokenAt(string& s, char sep, string t, int i);

*Descripción:* Reemplaza por t el token de s ubicado en la posición i.

*Parámetros:*

- string& s - Cadena tokenizada.
- char sep - Carácter separador.
- string t - Valor del nuevo token.
- int i - Posición del token que se será reemplazado por t.

*Retorna:* void.

Ejemplo de uso:

```
string s = "John|Paul|George|Ringo";
char sep = '|';
int i = 1;

string t = "McCartney";
setTokenAt(s, sep, t, i);
cout << s << endl; // Salida: John|McCartney|George|Ringo
```

#### 1.2.1.6. Función findToken

*Prototipo:* int findToken(string s, char sep, string t);

*Descripción:* Determinar la posición que el token  $t$  ocupa dentro de la cadena  $s$ .

Parámetros:

- `string s` - Cadena tokenizada.
- `char sep` - Carácter separador.
- `string t` - Token a buscar en la cadena  $s$ .

*Retorna:* `int` - La posición de la primera ocurrencia del token  $t$  dentro de la cadena  $s$ , o un valor negativo si  $s$  no contiene a  $t$ .

Ejemplo de uso:

```
string s = "John|Paul|George|Ringo";
char sep = '|';

string t = "Paul";
int p = findToken(s, sep, t);
cout << p << endl; // Salida: 1

string t = "John";
p = findToken(s, sep, t);
cout << p << endl; // Salida: 0
```

## 1.3. Lección 7

Comenzaremos analizando algunos ejemplos que ilustran la mayor parte de los casos de uso del TAD `Coll`. Luego pasaremos a las especificaciones.

### 1.3.1. TAD `Coll` (ejemplos de uso)

#### 1.3.1.1. Crear una colección, agregarle elementos e iterarla

```
Coll<string> c = coll<string>();
collAdd<string>(c, "John", stringToString);
collAdd<string>(c, "Paul", stringToString);
collAdd<string>(c, "George", stringToString);
collAdd<string>(c, "Ringo", stringToString);

collReset<string>(c);
while( collHasNext<string>(c) )
{
```

```

string s = collNext<string>(c, stringToString);
cout << s << endl;
}

```

#### 1.3.1.2. Iterar la colección, otra posibilidad

```

Coll<string> c = coll<string>();
collAdd<string>(c, "John", stringToString);
collAdd<string>(c, "Paul", stringToString);
collAdd<string>(c, "George", stringToString);
collAdd<string>(c, "Ringo", stringToString);

collReset<string>(c);
bool endOfColl;

string s = collNext<string>(c, endOfColl, stringToString);
while ( !endOfColl )
{
    cout << s << endl;
    s = collNext<string>(c, endOfColl, stringToString);
}

```

#### 1.3.1.3. Iterar la colección usando un ciclo for

```

Coll<string> c = coll<string>();
collAdd<string>(c, "John", stringToString);
collAdd<string>(c, "Paul", stringToString);
collAdd<string>(c, "George", stringToString);
collAdd<string>(c, "Ringo", stringToString);

for(int i=0; i<collSize<string>(c); i++)
{
    string s = collGetAt<string>(c, i, stringToString);
    cout << s << endl;
}

```

#### 1.3.1.4. Acceso directo a los elementos de la colección

```

Coll<string> c = coll<string>();

```

```

collAdd<string>(c, "John", stringToString);
collAdd<string>(c, "Paul", stringToString);
collAdd<string>(c, "George", stringToString);
collAdd<string>(c, "Ringo", stringToString);

int pos = 2;
string s = collGetAt<string>(c, pos, stringToString);
cout << s << endl; // George

```

#### 1.3.1.5. Reemplazar un elemento de la colección

```

Coll<string> c = coll<string>();
collAdd<string>(c, "John", stringToString);
collAdd<string>(c, "Paul", stringToString);
collAdd<string>(c, "George", stringToString);
collAdd<string>(c, "Ringo", stringToString);

int pos = 2;
string nuevo = "George Harrison";
collSetAt<string>(c, nuevo, pos, stringToString);

string s = collGetAt<string>(c, pos, stringToString);
cout << s << endl; // George Harrison

```

#### 1.3.1.6. Remove un elemento de la colección

```

Coll<string> c = coll<string>();
collAdd<string>(c, "John", stringToString);
collAdd<string>(c, "Paul", stringToString);
collAdd<string>(c, "George", stringToString);
collAdd<string>(c, "Ringo", stringToString);

int pos = 2;

// Salida: George
cout << collGetAt<string>(c, pos, stringToString) << endl;

collRemoveAt<string>(c, pos);

// Salida: Ringo
cout << collGetAt<string>(c, pos, stringToString) << endl;

```



**1.3.1.7. Buscar un elemento dentro de la colección**

```
struct Persona
{
    int dni; // documento nacional de identidad
    string nombre;
};
```

```
Coll<Persona> c = coll<Persona>();
collAdd<Persona>(c,persona(11,"Juan"),personaToString);
collAdd<Persona>(c,persona(44,"Pedro"),personaToString);
collAdd<Persona>(c,persona(33,"Carlos"),personaToString);
collAdd<Persona>(c,persona(22,"Pablo"),personaToString);

int dni=33;
int pos = collFind<Persona,int>(c
                                ,dni
                                ,cmpPersonaDNI
                                ,personaFromString);

Persona p = collGetAt<Persona>(c,pos,personaFromString);
cout << personaToString(p) << endl;
```

```
int cmpPersonaDNI(Persona p,int dni)
{
    return p.dni-dni;
}
```

**1.3.1.8. Ordenar una colección**

```
struct Persona
{
    int dni; // documento nacional de identidad
    string nombre;
};
```

```

Coll<Persona> c = coll<Persona>();
collAdd<Persona>(c, persona(11, "Juan"), personaToString);
collAdd<Persona>(c, persona(44, "Pedro"), personaToString);
collAdd<Persona>(c, persona(33, "Carlos"), personaToString);
collAdd<Persona>(c, persona(22, "Pablo"), personaToString);

// ordenamos por nombre alfabeticamente
collSort<Persona>(c
                  , cmpPersonaNombre
                  , personaFromString
                  , personaToString);

// iteramos y mostramos
mostrarColeccion(c);

// ordenamos por DNI ascendente
collSort<Persona>(c
                  , cmpPersonaDNI
                  , personaFromString
                  , personaToString);

// iteramos y mostramos
mostrarColeccion(c);

```

```

void mostrarColeccion(Coll<Persona> c)
{
    collReset<Persona>(c);
    while( collHasNext<Persona>(c) )
    {
        Persona p = collNext<Persona>(c, personaFromString);
        cout << personaToString(p) << endl;
    }
}

```

```

int cmpPersonaNombre(Persona p, String nombre)
{
    return cmpString(p.nombre, nombre);
}

```

```
int cmpPersonaDNI(Persona p,int dni)
{
    return p.dni-dni;
}
```

### 1.3.2. TAD Coll (API)

La siguientes funciones deben programarse en el archivo `Coll.hpp`.

#### 1.3.2.1. Estructura del TAD

```
template<typename T>
struct Coll
{
    // implementacion a cargo del estudiante
};
```

#### 1.3.2.2. Función coll

*Prototipo:* `Coll<T> coll(char sep);`

*Descripción:* Crea una colección vacía, preparada para contener elementos de tipo `T`; utilizando el carácter `sep` como separador de la cadena *tokenizada* sobre la que se implementa la colección.

*Parámetro:* `char sep` – Carácter separador.

*Retorna:* `Coll<T>` - Una colección vacía preparada para contener elementos tipo `T`.

#### 1.3.2.3. Función coll (sobrecarga)

*Prototipo:* `Coll<T> coll();`

*Descripción:* Crea una colección vacía, preparada para contener elementos tipo `T`; definiendo un separador por defecto para usar en la cadena *tokenizada* sobre la cual se implementa la colección.

*Retorna:* `Coll<T>` - Una colección vacía preparada para contener elementos tipo `T`.

#### 1.3.2.4. Función collSize

*Prototipo:* `int collSize(Coll<T> c);`

*Descripción:* Retorna la cantidad de elementos que contiene la colección `c`.

*Parámetro:* `Coll<T> c` - Colección para determinar cuántos elementos tiene.

*Retorna:* `int` - Cantidad de elementos que tiene la colección `c`.

#### **1.3.2.5. Función `collRemoveAll`**

*Prototipo:* `void collRemoveAll(Coll<T>& c);`

*Descripción:* Remueve de la colección `c` todos sus elementos, dejándola vacía.

*Parámetro:* `Coll<T>& c` - Colección cuyos elementos serán removidos.

*Retorna:* `void`.

#### **1.3.2.6. Función `collRemoveAt`**

*Prototipo:* `void collRemoveAt(Coll<T>& c, int p);`

*Descripción:* Remueve de la colección `c` el elemento ubicado en la posición `p`.

*Parámetros:*

- `Coll<T>& c` - Colección de la cual se eliminará un elemento.
- `int p` - Posición del elemento que se eliminará.

*Retorna:* `void`.

#### **1.3.2.7. Función `collAdd`**

*Prototipo:* `int collAdd(Coll<T>& c, T t, string tToString(T));`

*Descripción:* Agrega el elemento `t` al final de la colección `c`.

*Parámetros:*

- `Coll<T>& c` - La colección.
- `T t` - Elemento que se va a agregar al final de `c`.
- `string tToString(T)` - Función que convierte de `T` a `string`.

*Retorna:* `int` - La posición que ocupa el elemento recientemente agregado. Coincide con el tamaño de la colección, menos 1.

**1.3.2.8. Función collSetAt**

*Prototipo:* `void collSetAt(Coll<T>& c  
                          , T t  
                          , int p  
                          , string tToString(T));`

*Descripción:* Reemplaza por `t` al elemento que se ubica en la posición `p`.

*Parámetros:*

- `Coll<T>& c` - La colección.
- `T t` - Elemento que se asignará en la posición `p`.
- `int p` - Posición donde quedará asignado `t`.
- `string tToString(T)` - Función que convierte de `T` a `string`.

*Retorna:* `void`.

**1.3.2.9. Función collGetAt**

*Prototipo:* `T collGetAt(Coll<T> c, int p, T tFromString(string));`

*Descripción:* Retorna el elemento que se ubica en la posición `p` de la colección `c`.

*Parámetros:*

- `Coll<T> c` - La colección.
- `int p` - Posición del elemento al que se quiere acceder.
- `T tFromString(string)` - Función que convierte de `string` a `T`.

*Retorna:* `T` - El elemento de `c` ubicado en la posición `p`.

**1.3.2.10. Función collFind**

*Prototipo:* `int collFind(Coll<T> c  
                          , K k  
                          , int cmpTK(T, K)  
                          , T tFromString(string));`

*Descripción:* Determina si la colección `c` contiene al elemento `k`.

*Parámetros:*

- `Coll<T> c` - La colección.
- `K k` - Elemento que se debe buscar dentro de `c`.

- `int cmpTK(T,K)` - Función que compara un elemento tipo `T` (`t`) con otro tipo `K` (`k`) y retorna: negativo si  $t < k$ ; cero si  $t == k$  o positivo si  $t > k$ .
- `T tFromString(string)` - Función que convierte de `string` a `T`.

**Retorna:** `int` - La posición que ocupa la primera ocurrencia de `k` dentro de `c` o un valor negativo si `c` no contiene a `k`.

Ejemplo de uso:

### 1.3.2.11. Función collSort

```
Prototipo: void collSort(Coll<T>& c
                        ,int cmpTT(T,T)
                        ,T tFromString(string)
                        ,string tToString(T));
```

*Descripción:* Ordena los elementos de la colección  $c$  según el criterio de precedencia que establece  $\text{cmpTT}$ .

*Parámetros:*

- `Coll<T>& c` - La colección.
- `int cmpTT(T, T)` - Función que compara dos elementos tipo `T` (`t1,t2`) y retorna: negativo si `t1<t2`; cero si `t1==t2`; positivo si `t1>t2`.
- `T tFromString(string)` - Función que convierte de `string` a `T`.
- `string tToString(T)` - Función que convierte de `T` a `string`.

*Retorna:* void.

### 1.3.2.12. Función collHasNext

**Prototipo:** `bool collHasNext (Coll<T> c);`

**Descripción:** Retorna `true` o `false` según queden, en la colección `c`, más elementos para continuar iterando.

**Parámetro:** Coll<T> c – La colección.

**Retorna:** bool - true o false según queden o no elementos para continuar iterando sobre la colección c.

**1.3.2.13. Función collNext**

*Prototipo:* `T collNext(Coll<T>& c, T tFromString(string));`

*Descripción:* Retorna el próximo elemento de la colección `c`.

*Parámetros:*

- `Coll<T>& c` - La colección.
- `T tFromString(string)` - Función que convierte de `string` a `T`.

*Retorna:* `T` - El siguiente elemento de la colección `c`.

**1.3.2.14. Función collNext (sobrecarga)**

*Prototipo:* `T collNext(Coll<T>& c, bool& eoc, T tFromString(string));`

*Descripción:* Retorna el próximo elemento de la colección `c`, indicando si se llegó al final de la colección. De este modo, permite prescindir de usar `collHasNext`.

*Parámetros:*

- `Coll<T>& c` - La colección.
- `bool& eoc` - Variable para indicar si se llegó al final de la colección.
- `T tFromString(string)` - Función que convierte de `string` a `T`.

*Retorna:* `T` - El siguiente elemento de la colección `c`.

**1.3.2.15. Función collReset**

*Prototipo:* `void collReset(Coll<T>& c);`

*Descripción:* Reinicia la colección `c` para que la podamos volver a iterar.

*Parámetro:* `Coll<T>& c` - La colección.

*Retorna:* `void`.

## 1.4. Lección 10

**1.4.1. API de tratamiento de archivos de registros**

La siguientes funciones deben programarse en el archivo `files.hpp`.

**1.4.1.1. Función write**

*Prototipo:* `void write(FILE* f, T t);`

*Descripción:* Escribe el valor `t` en la posición actual del archivo `f`.

*Parámetros:*

- `FILE* f` – Archivo donde vamos a escribir.
- `T` – valor (registro) que vamos a escribir en `f`.

*Retorna:* `void`.

Ejemplo de uso:

```
FILE* f = fopen("numeros.x", "w+b");
write<short>(f, 1234);
write<short>(f, 4321);
write<short>(f, -9876);

fclose(f);
```

**1.4.1.2. Función read**

*Prototipo:* `T read(FILE* f);`

*Descripción:* Lee del archivo `f` un registro tipo `T` y retorna el valor leído.

*Parámetro:* `FILE* f` – Archivo desde el cual vamos a leer un registro.

*Retorna:* `T` – Registro leído.

Ejemplo de uso:

```
FILE* f = fopen("numeros.x", "w+b");

short s = read<short>(f);
while( !feof(f) )
{
    cout << s << endl;
    short s = read<short>(f);
}

fclose(f);
```



**1.4.1.3. Función seek**

*Prototipo:* `void seek(FILE* f, int n);`

*Descripción:* Mueve el indicador de posición del archivo `f` al inicio del registro `n`.

*Parámetros:*

- `FILE* f` – Archivo cuyo indicador de posición vamos a modificar.
- `int n` – Número de registro al que haremos apuntar el indicador de posición.

*Retorna:* `void`.

*Ejemplo de uso:*

```
FILE* f = fopen("numeros.x", "w+b");

// apunto al tercer registro (comenzando desde cero)
seek<short>(f, 2);

// leo el registro apuntado por el indicador de posicion
short v = read<short>(f);
```

**1.4.1.4. Función fileSize**

*Prototipo:* `int fileSize(FILE* f);`

*Descripción:* Retorna la cantidad de registros tipo `T` que contiene el archivo.

*Parámetro:* `FILE* f` – Archivo.

*Retorna:* `int` – Cantidad de registros tipo `T` que contiene el archivo `f`.

*Ejemplo de uso:*

```
FILE* f = fopen("numeros.x", "w+b");

// mostramos el archivo desde el final hasta el inicio
for(int i=fileSize<short>(f)-1; i>=0; i--)
{
    seek<short>(f, i);
    short s = read<short>(f);
    cout << s << endl;
```

```

}

fclose(f);

```

#### 1.4.1.5. Función filePos

**Prototipo:** `int filePos(FILE* f);`

**Descripción:** Retorna el número de registro que está siendo apuntado por el indicador de posición del archivo `f`.

**Parámetro:** `FILE* f` – Archivo.

**Retorna:** `int` – Número de registro apuntado por el indicador de posición.

Ejemplo de uso:

```

FILE* f = fopen("numeros.x", "w+b");

// mostramos el archivo desde el final hasta el inicio
for( int i=fileSize<short>(f)-1; i>=0; i-- )
{
    seek<short>(f,i);

    // numero de registro apuntado por el indicador de posicion
    int pos = filePos<short>(f);

    short s = read<short>(f);
    cout << "Registro Nro. " << pos << ", " << s << endl;
}

fclose(f);

```

## 1.5. Lección 12

### 1.5.1. API de tratamiento de arrays

La siguientes funciones deben programarse en el archivo `arrays.hpp`.

#### 1.5.1.1. Función add

**Prototipo:** `int add(T arr[], int& len, T e);`

*Descripción:* Agrega el elemento `e` al final de `arr` incrementando su longitud `len`.

*Parámetros:*

- `T arr[]` – Array donde agregaremos un elemento.
- `int& len` – Longitud actual del array.
- `T e` – Elemento que vamos a agregar.

*Retorna:* `int` – Posición del array donde quedó ubicado el elemento que agregamos.

Ejemplo de uso:

```
// array y longitud
string a[10];
int len = 0;

// agrego elementos
add<string>(a, len, "John");
add<string>(a, len, "Paul");
add<string>(a, len, "George");
add<string>(a, len, "Ringo");

// recorro y muestro
for(int i=0; i<len; i++)
{
    cout << a[i] << endl;
}
```

### 1.5.1.2. Función insert

*Prototipo:* `void insert(T arr[], int& len, T e, int p);`

*Descripción:* Inserta el elemento `e` en la posición `p` del array `arr`. Desplaza los elementos ubicados a partir de `p+1` e incrementa la longitud `len`.

*Parámetros:*

- `T arr[]` – Array donde insertaremos un elemento.
- `int& len` – Longitud actual del array.
- `T e` – Elemento que vamos a agregar.
- `int p` – Posición donde se insertará el nuevo elemento.

*Retorna:* `void`.

Ejemplo de uso:

```
// array y longitud
string a[10];
int len = 0;

// agrego elementos
insert<string>(a, len, "John", 0);
insert<string>(a, len, "Paul", 0);
insert<string>(a, len, "George", 0);
insert<string>(a, len, "Ringo", 0);

// recorro y muestro
for(int i=0; i<len; i++)
{
    cout << a[i] << endl; // SALIDA: Ringo, George, Paul, John
}
```

#### 1.5.1.3. Función remove

**Prototipo:** `T remove(T arr[], int& len, int p);`

**Descripción:** Remueve el elemento ubicado en la posición `p` del array `arr`. Desplaza ubicados a partir de `p` y decrementa la longitud `len`.

**Parámetros:**

- `T arr[]` – Array donde removeremos un elemento.
- `int& len` – Longitud actual del array.
- `T p` – Posición cuyo elemento será removido.

**Retorna:** `T` – Elemento que fue removido del array.

Ejemplo de uso:

```
// array y longitud
string a[10];
int len = 0;

// agrego elementos
add<string>(a, len, "John");
add<string>(a, len, "Paul");
```

```

add<string>(a, len, "George");
add<string>(a, len, "Ringo");

while( len>0 )
{
    cout << remove<string>(arr, len, 0) << endl;
}

```

#### 1.5.1.4. Función find

*Prototipo:* `int find(T arr[], int len, K k, int cmpTK(T, K));`

*Descripción:* Retorna la posición de la primera ocurrencia de `k` dentro de `arr` o un valor negativo si `arr` no contiene a `k`.

*Parámetros:*

- `T arr[]` – Array donde buscaremos un elemento.
- `int len` – Longitud actual del array.
- `K k` – Valor a buscar dentro de `arr`.
- `int cmpTK(T, K)` – Función de comparación.

*Retorna:* `int` – Posición de la primera ocurrencia de `k` dentro de `arr` o un valor negativo si `arr` no contiene a `k`.

Ejemplo de uso:

```

struct Persona
{
    int dni;
    string nom;
};

```

```

int cmpPersonaDNI(Persona p, int d)
{
    return p.dni-d;
}

```

```
// array de personas
int len=3;
Persona arr[] = {persona(10,"Pablo")
                 ,persona(20,"Pedro")
                 ,persona(30,"Juan")};

// busco por DNI
int pos = find<Persona,int>(arr,len,20,cmpPersonaDNI);
cout << pos << endl; // SALIDA: 1
```

#### 1.5.1.5. Función orderedInsert

**Prototipo:** `int orderedInsert(T arr[],int& len,T e,int cmpTT(T,T));`

**Descripción:** Inserta `e` dentro de `arr` según el criterio de precedencia que establece `cmpTT`, y retorna la posición donde dicho elemento quedó insertado. El `array arr` debe estar ordenado o vacío.

**Parámetros:**

- `T arr[]` - Array donde insertaremos un elemento.
- `int& len` - Longitud actual del `array`.
- `T e` - Valor a insertar dentro de `arr`.
- `int cmpTT(T,T)` - Función de comparación.

**Retorna:** `int` - Posición donde quedó insertado `e` dentro de `arr`.

Ejemplo de uso:

```
// funcion de comparacion
int cmpInt(int a,int b){return a-b;}

// array y longitud
int arr[10] = {1,2,3,5,6,7,8};
int len = 7;

int pos = orderedInsert<int>(arr,len,4,cmpInt);
cout << pos << endl;
```

#### 1.5.1.6. Función sort

**Prototipo:** `void sort(T arr[],int len,int cmpTT(T,T));`

*Descripción:* Ordena `arr` según el criterio de precedencia que establece `cmpTT`.

*Parámetros:*

- `T arr[]` – Array que ordenaremos.
- `int len` – Longitud actual del array.
- `int cmpTT(T, T)` – Función de comparación.

*Retorna:* `void`.

Ejemplo de uso:

```
// array y longitud
int arr[] = {5,4,3,2,1};
int len = 5;

// ordeno y muestro
sort<int>(arr, len, cmpInt);
for(int i=0; i<len; i++)
{
    cout << arr[i] << endl;
}
```

```
// funcion de comparacion
int cmpInt(int a, int b) {return a-b;}
```

## 1.6. Lección 13

### 1.6.1. TAD Array

Las siguientes funciones deben programarse en el archivo `Array.hpp`.

#### 1.6.1.1. Estructura del TAD

```
template<typename T>
struct Array
{
```

```
// implementacion a cargo del estudiante
};
```

### 1.6.1.2. Función array

*Prototipo:* `Array<T> array();`

*Descripción:* Inicializa un *array* cuya capacidad inicial será establecida por defecto. La longitud del *array* será 0, y se incrementará a medida que se agreguen o inserten nuevos elementos.

*Retorna:* `Array<T>` – El *array*.

Ejemplo de uso:

```
// array y longitud
Array<int> a = array<int>();
arrayAdd<int>(a,10);
arrayAdd<int>(a,20);
arrayAdd<int>(a,30);

// el size del array?
cout << arraySize<int>(arr) << endl; // Salida: 3
```

### 1.6.1.3. Función arrayAdd

*Prototipo:* `int arrayAdd(Array<T>& a, T t);`

*Descripción:* Agrega *t* al final de *a* incrementando, de ser necesario, su capacidad. Retorna la posición del *arr* donde quedó ubicado el elemento *t*.

*Parámetros:*

- `Array<T>& a` – El *array*.
- `T t` – Elemento que se agregará.

*Retorna:* `int` – Posición de *a* donde se agregó el elemento *t*.

Ejemplo de uso:

```
Array<int> a = array<int>();
int pos = arrayAdd<int>(a,10); // pos = 0
```



**1.6.1.4. Función arrayGet**

*Prototipo:* `T* arrayGet(Array<T> a, int p);`

*Descripción:* Retorna la dirección del elemento de `a` ubicado en la posición `p`.

*Parámetros:*

- `Array<T> a` – El array.
- `int p` – Posición del elemento de `a` al cual queremos acceder.

*Retorna:* `T*` – Dirección del elemento ubicado en la posición `p` del array `a`.

Ejemplo de uso:

```
Array<int> a = array<int>();
arrayAdd<int>(a, 10);
arrayAdd<int>(a, 20);
arrayAdd<int>(a, 30);

int* p = arrayGet<int>(a, 1);
*p = 22; // cambia 20 por 22
```

**1.6.1.5. Función arraySet**

*Prototipo:* `void arraySet(Array<T>& a, int p, T t);`

*Descripción:* Asigna el elemento `t` en la posición `p` del array `a`.

*Parámetros:*

- `Array<T>& a` – El array.
- `int p` – Posición del elemento de `a` al cual queremos acceder.
- `T t` – Elemento que vamos a asignar en la posición `p` de `a`.

*Retorna:* `void`.

Ejemplo de uso:

```
Array<int> a = array<int>();
arrayAdd<int>(a, 10);
arrayAdd<int>(a, 20);
arrayAdd<int>(a, 30);
arraySet<int>(a, 1, 99); // reemplaza 20 x 99
```

**1.6.1.6. Función arrayInsert**

*Prototipo:* `void arrayInsert(Array<T>& a, T t, int p);`

*Descripción:* Inserta `t` en la posición `p` del array `a`.

*Parámetros:*

- `Array<T>& a` – El array.
- `T t` – Elemento a insertar.
- `int p` – Posición donde quedará insertado `t`.

*Retorna:* `void`.

Ejemplo de uso:

```
Array<int> a = array<int>();
arrayInsert<int>(a, 10, 0);
arrayInsert<int>(a, 20, 0);
arrayInsert<int>(a, 30, 0);

int* p = arrayGet<int>(a, 0);
cout << *p << endl; // SALIDA: 30
```

**1.6.1.7. Función arraySize**

*Prototipo:* `int arraySize(Array<T> a);`

*Descripción:* Retorna la longitud actual del array.

*Parámetro:* `Array<T> a` – El array.

*Retorna:* `int` – Longitud del array `a`.

Ejemplo de uso:

```
Array<int> a = array<int>();
arrayAdd<int>(a, 10);
arrayAdd<int>(a, 20);
arrayAdd<int>(a, 30);

for(int i=0; i<arraySize<int>(a); i++)
{
    int* e = arrayGet<int>(a, i)
```

```
    cout << *e << endl; // SALIDA: 10,20,30
}
```

#### 1.6.1.8. Función arrayRemove

*Prototipo:* T arrayRemove(Array<T>& a, int p);

*Descripción:* Remove el elemento de a ubicado en la posición p.

*Parámetros:*

- Array<T>& a - El array.
- int p - Posición a remover.

*Retorna:* T - Elemento que ocupaba la posición p dentro de a.

*Ejemplo de uso:*

```
Array<int> a = array<int>();
arrayAdd<int>(a,10);
arrayAdd<int>(a,20);
arrayAdd<int>(a,30);

int e = arrayRemove<int>(a,0);
cout << e << endl; // SALIDA: 10
```

#### 1.6.1.9. Función arrayRemoveAll

*Prototipo:* void arrayRemoveAll(Array<T>& a);

*Descripción:* Remueve todos los elemento de a dejándolo vacío, con longitud 0.

*Parámetro:* Array<T>& a - El array.

*Retorna:* void.

*Ejemplo de uso:*

```
Array<int> a = array<int>();
arrayAdd<int>(a,10);
arrayAdd<int>(a,20);
arrayAdd<int>(a,30);
```

```
// elimino todos los elementos
arrayRemoveAll<int>(a);

cout << arraySize<int>(a) << endl; // Salida: 0
```

#### 1.6.1.10. Función arrayFind

**Prototipo:** `int arrayFind(Array<T> a, K k, int cmpTK(T, K)) ;`

**Descripción:** Retorna la posición que `k` ocupa dentro de `a`, según la función de comparación `cmpTK`, o un valor negativo si `a` no contiene a `k`.

**Parámetros:**

- `Array<T>& a` – El array.
- `int k` – Elemento a buscar.
- `int cmpTK(T, K)` – Función de comparación.

**Retorna:** `int` – Posición de la primera ocurrencia de `k` dentro de `a` o un valor negativo si `a` no contiene a `k`.

Ejemplo de uso:

```
Array<int> a = array<int>();
arrayAdd<int>(a, 10);
arrayAdd<int>(a, 20);
arrayAdd<int>(a, 30);

int pos = arrayFind<int, int>(a, 30, cmpInt);
cout << pos << endl; // SALIDA: 2
```

#### 1.6.1.11. Función arrayOrderedInsert

**Prototipo:** `int arrayOrderedInsert(Array<T>& a, T t, int cmpTT(T, T)) ;`

**Descripción:** Inserta `t` en `a` según el criterio de precedencia que establece `cmpTT`.

**Parámetros:**

- `Array<T>& a` – El array.
- `T t` – Elemento a insertar.
- `int cmpTT(T, T)` – Función de comparación.

*Retorna:* `int` – Posición donde quedó insertado `t` dentro de `a`.

Ejemplo de uso:

```
Array<int> a = array<int>();
arrayOrderedInsert<int>(a, 2, cmpInt);
arrayOrderedInsert<int>(a, 1, cmpInt);
arrayOrderedInsert<int>(a, 3, cmpInt);

for(int i=0; i<arraySize<int>(a); i++)
{
    int* p = arrayGet<int>(a, i);
    cout << *p << endl; // SALIDA 1,2,3
}
```

#### 1.6.1.12. Función `arraySort`

*Prototipo:* `void arraySort(Array<T>& a, int cmpTT(T, T));`

*Descripción:* Ordena el array `a` según establece `cmpTT`.

*Parámetros:*

- `Array<T>& a` – El array.
- `int cmpTT(T, T)` – Función de comparación.

*Retorna:* `void`.

Ejemplo de uso:

```
Array<int> a = array<int>();
arrayAdd<int>(a, 2);
arrayAdd<int>(a, 1);
arrayAdd<int>(a, 3);

// ordeno
arraySort<int>(a, cmpInt)
```

### 1.6.2. TAD Map

Las siguientes funciones deben programarse en el archivo `Map.hpp`.

**1.6.2.1. Estructura del TAD**

```
template<typename K, template V>
struct Map
{
    // implementacion a cargo del estudiante
};
```

**NOTA:** El tipo de dato de la clave (K) será primitivo, `string` o cualquier otro cuya implementación soporte el uso de los operadores relacionales `<`, `>`, `==` y `!=`.

El TAD `Map` debe implementarse usando el TAD `Array`.

**1.6.2.2. Función map**

*Prototipo:* `Map<K,V> map () ;`

*Descripción:* Inicializa un `map`.

*Retorna:* `Map<K,V>` – El `map`.

Ejemplo de uso:

```
Map<string,int> m = map<string,int>();
mapPut<string,int>(m,"uno",1);
mapPut<string,int>(m,"dos",2);
mapPut<string,int>(m,"tres",3);

int* n = mapGet<string,int>(m,"dos");
cout << *n << endl; // Salida: 2

string k = "uno";
if( mapContains<string,int>(m,k) )
{
    cout << "Existe una entrada con clave" << k << endl;
}
```

**1.6.2.3. Función mapGet**

*Prototipo:* `V* mapGet (Map<K,V> m, K k) ;`

*Descripción:* Retorna la dirección de memoria del valor asociado a la clave `k` o `NULL` si `m` no contiene ningún valor asociado a dicha clave.

*Parámetros:*

- $\text{Map}<K, V> \ m$  – El *map*.
- $K \ k$  – Clave con la cual, dentro del *map*, quedará asociado el elemento  $v$ .

*Retorna:*  $V^*$  - Dirección de memoria del elemento vinculado con la clave  $k$  o `NULL` si  $m$  no contiene ningún valor asociado a  $k$ .

Ejemplo de uso: ver anterior.

#### **1.6.2.4. Función mapPut**

*Prototipo:*  $V^* \text{ mapPut } (\text{Map}<K, V>\& \ m, K \ k, V \ v) ;$

*Descripción:* Agrega al *map*  $m$  el elemento  $v$  asociado a la clave  $k$ . Si existía una entrada vinculada a  $k$  se debe reemplazar el valor anterior por  $v$ .

*Parámetros:*

- $\text{Map}<K, V>\& \ m$  – El *map*.
- $K \ k$  – Clave con la cual, dentro del *map*, quedará asociado el elemento  $v$ .
- $V \ v$  – Valor o elemento a agregar.

*Retorna:*  $V^*$  - Dirección de memoria del elemento vinculado con la clave  $k$ .

Ejemplo de uso: Ver anterior.

#### **1.6.2.5. Función mapContains**

*Prototipo:*  $\text{bool mapContains } (\text{Map}<K, V> \ m, K \ k) ;$

*Descripción:* Verifica si  $m$  contiene a  $k$ .

*Parámetros:*

- $\text{Map}<K, V> \ m$  – El *map*.
- $K \ k$  – Clave.

*Retorna:* `bool` - `true` o `false` según  $m$  contenga, o no, una entrada vinculada a  $k$ .

Ejemplo de uso: Ver anterior.

#### **1.6.2.6. Función mapRemove**

*Prototipo:*  $V \text{ mapRemove } (\text{Map}<K, V>\& \ m, K \ k) ;$

*Descripción:* Elimina de  $m$  la entrada identificada con la clave  $k$ .

*Parámetros:*

- $\text{Map}<K, V>\& m$  – El *map*.
- $K\ k$  – Clave que identifica la entrada a remover.

*Retorna:*  $V$  – Valor que contenía la entrada asociada a la clave  $k$ .

#### 1.6.2.7. Función `mapRemoveAll`

*Prototipo:* `void mapRemoveAll (Map<K, V>& m) ;`

*Descripción:* Elimina todas las entradas del *map*  $m$ .

*Parámetro:*  $\text{Map}<K, V>\& m$  – El *map*.

*Retorna:* `void`.

#### 1.6.2.8. Función `mapSize`

*Prototipo:* `int mapSize (Map<K, V> m) ;`

*Descripción:* Retorna la cantidad actual de entradas que tiene  $m$ .

*Parámetro:*  $\text{Map}<K, V>\& m$  – El *map*.

*Retorna:* `int` – Cantidad de entradas que tiene el *map*  $m$ .

#### 1.6.2.9. Función `mapHasNext`

*Prototipo:* `bool mapHasNext (Map<K, V> m) ;`

*Descripción:* Indica si quedan más elementos para continuar iterando el *map*.

*Parámetro:*  $\text{Map}<K, V> m$  – El *map*.

*Retorna:* `bool` – `true` o `false` según queden elementos para continuar iterando.

Ejemplo de uso:

```
Map<string, int> m = map<string, int>();
mapPut<string, int>(m, "uno", 1);
mapPut<string, int>(m, "dos", 2);
mapPut<string, int>(m, "tres", 3);
mapPut<string, int>(m, "cuatro", 4);
```



```

// iteramos el map accediendo a cada key
mapReset<string,int>(m);
while( mapHasNext<string,int>(m) )
{
    // clave y valor
    string k = mapNextKey<string,int>(m);
    int* v = mapGet<string,int>(m,k);
    cout << k << ", " << *v << endl;
}

// iteramos el map accediendo a cada value
mapReset<string,int>(m);
while( mapHasNext<string,int>(m) )
{
    // valor
    int* v = mapNextValue<string,int>(m);
    cout << *v << endl;
}

```

#### 1.6.2.10. Función mapNextKey

*Prototipo:* K mapNextKey (Map<K,V>& m);

*Descripción:* Permite iterar sobre las claves del *map*. Esta función es mutuamente excluyente respecto de mapNextValue.

*Parámetro:* Map<K,V>& m – El *map*.

*Retorna:* K – La siguiente clave dentro de una iteración.

Ejemplo de uso: Ver anterior.

#### 1.6.2.11. Función mapNextValue

*Prototipo:* V\* mapNextValue (Map<K,V>& m);

*Descripción:* Permite iterar sobre los valores que contiene el *map*. Esta función es mutuamente excluyente respecto de mapNextKey.

*Parámetro:* Map<K,V>& m – El *map*.

*Retorna:* V\* – Dirección de memoria del siguiente valor dentro de una iteración.

Ejemplo de uso: Ver anterior.

**1.6.2.12. Función mapReset**

*Prototipo:* `void mapReset (Map<K,V>& m) ;`

*Descripción:* Prepara el *map* para comenzar una nueva iteración.

*Parámetro:* `Map<K,V>& m` – El *map*.

*Retorna:* `void`.

Ejemplo de uso: Ver anterior.

**1.6.2.13. Función mapSortByKey**

*Prototipo:* `void mapSortByKey (Map<K,V>& m, int cmpKK (K,K) ) ;`

*Descripción:* Ordena el *map* aplicando sobre sus claves el criterio que establece `cmpKK`.

*Parámetros:*

- `Map<K,V>& m` – El *map*.
- `int cmpKK (K,K)` - Función de comparación.

*Retorna:* `void`.

**1.6.2.14. Función mapSortByValues**

*Prototipo:* `void mapSortByValues (Map<K,V>& m, int cmpVV (V,V) ) ;`

*Descripción:* Ordena el *map* aplicando sobre sus *values* el criterio que establece `cmpVV`.

*Parámetros:*

- `Map<K,V>& m` – El *map*.
- `int cmpVV (V,V)` – Función de comparación.

*Retorna:* `void`.

## 1.7. Lección 14

**1.7.1. API de tratamiento de listas enlazadas**

La siguientes funciones deben programarse en el archivo `lists.hpp`.

**1.7.1.1. Nodo**

```
template<typename T>
struct Node
{
    T info;
    Node<T>* sig;
};
```

Todos los ejemplos de uso, siempre que sea necesario invocar a una función de comparación, invocaremos a `cmpInt` cuyo código es el siguiente.

```
int cmpInt(int a,int b){return a-b;}
```

**1.7.1.2. Función add**

*Prototipo:* `Node<T>* add(Node<T>* &p, T e);`

*Descripción:* Agrega el elemento `e` al final de la lista direccionada por `p`.

*Parámetros:*

- `Node<T>* &p` – Puntero al primer nodo de la lista.
- `T e` – Elemento que vamos a agregar.

*Retorna:* `Node<T>*` – Dirección del nodo que contiene al elemento que se agregó.

Ejemplo de uso:

```
Node<int>* p = NULL;
add<int>(p,1);
add<int>(p,2);
add<int>(p,3); // p->{1,2,3}
```

**1.7.1.3. Función addFirst**

*Prototipo:* `Node<T>* addFirst(Node<T>* &p, T e);`

*Descripción:* Agrega el elemento `e` al inicio de la lista direccionada por `p`.

*Parámetros:*

- `Node<T>*& p` – Puntero al primer nodo de la lista.
- `T e` – Elemento que vamos a agregar al inicio de la lista.

*Retorna:* `Node<T>*` – Dirección del nodo que contiene al elemento que se agregó.

Ejemplo de uso:

```
Node<int>* p = NULL;
addFirst<int>(p,1);
addFirst<int>(p,2);
addFirst<int>(p,3); // p->{3,2,1}
```

#### 1.7.1.4. Función remove

*Prototipo:* `T remove(Node<T>*& p, K k, int cmpTK(T, K))`;

*Descripción:* Remueve la primera ocurrencia del elemento concordante con `cmpTK`.

*Parámetros:*

- `Node<T>*& p` – Puntero al primer nodo de la lista.
- `K k` – Elemento o clave de búsqueda del elemento que vamos a remover.
- `int cmpTK(T, K)` – Función de comparación.

*Retorna:* `T` – Valor del elemento que fue removido.

Ejemplo de uso:

```
Node<int>* p = NULL;
add<int>(p,1);
add<int>(p,2);
add<int>(p,3); // p->{1,2,3}

int e = remove<int,int>(p,2,cmpInt); // p->{1,3}
cout << e << endl; // Salida: 2
```

#### 1.7.1.5. Función removeFirst

*Prototipo:* `T removeFirst(Node<T>*& p)`;

*Descripción:* Remueve el primer elemento de la lista direccionada por `p`.

*Parámetro:* `Node<T>*& p` – Puntero al primer nodo de la lista.

*Retorna:*  $T$  – Valor del elemento que acabamos de remover.

Ejemplo de uso:

```
Node<int>* p = NULL;
add<int>(p,1);
add<int>(p,2);
add<int>(p,3); // p->{1,2,3}

int e = removeFirst(p); // p->{2,3}
cout << e << endl;    // Salida: 1
```

#### 1.7.1.6. Función find

*Prototipo:* `Node<T>* find(Node<T>* p, K k, int cmpTK(T, K))`;

*Descripción:* Retorna la dirección del nodo que contiene la primera ocurrencia de  $k$ , según `cmpTK`, o `NULL` si ningún elemento concuerda con dicha clave de búsqueda.

*Parámetros:*

- `Node<T>* p` – Puntero al primer nodo de la lista.
- `K k` – Elemento o clave de búsqueda del elemento.
- `int cmpTK(T, K)` – Función de comparación.

*Retorna:* `Node<T>*` – Dirección del nodo que contiene la primera ocurrencia del elemento que buscamos o `NULL` si la lista no contiene dicho elemento.

Ejemplo de uso:

```
Node<int>* p = NULL;
add<int>(p,1);
add<int>(p,2);
add<int>(p,3); // p->{1,2,3}

Node<int>* e = find<int,int>(p,2,cmpInt);
cout << e->info << endl; // Salida: 2
```

#### 1.7.1.7. Función orderedInsert

*Prototipo:* `Node<T>* orderedInsert(Node<T>* &p, T e, int cmpTT(T, T))`;

*Descripción:* Inserta el elemento  $e$  en la lista direccionada por  $p$  según el criterio que establece la función  $\text{cmpTT}$ . La lista debe estar vacía u ordenada según  $\text{cmpTT}$ .

*Parámetros:*

- $\text{Node}<\text{T}>^* \& p$  – Puntero al primer nodo de la lista.
- $\text{T } e$  – Elemento que vamos a insertar.
- $\text{int } \text{cmpTT}(\text{T}, \text{T})$  – Función que establece el criterio de ordenamiento.

*Retorna:*  $\text{Node}<\text{T}>^*$  – Dirección del nodo que acabamos de insertar.

Ejemplo de uso:

```
Node<int>* p = NULL;
orderedInsert<int>(p, 2, cmpInt);
orderedInsert<int>(p, 3, cmpInt);
orderedInsert<int>(p, 1, cmpInt); // p->{1,2,3}
```

#### 1.7.1.8. Función searchAndInsert

*Prototipo:*  $\text{Node}<\text{T}>^* \text{ searchAndInsert}(\text{Node}<\text{T}>^* \& p$   
 $\text{, T } e$   
 $\text{, bool\& enc}$   
 $\text{, int } \text{cmpTT}(\text{T}, \text{T}))$ ;

*Descripción:* Busca en la lista direccionada por  $p$  la primera ocurrencia de  $e$ , y retorna la dirección del nodo que lo contiene. Si  $e$  no existe en la lista entonces lo inserta en orden, según el criterio establecido por  $\text{cmpTT}$ , y retorna la dirección del nodo insertado. Asigna `true` o `false` a `enc` según  $e$  fue encontrado o insertado.

*Parámetros:*

- $\text{Node}<\text{T}>^* \& p$  – Puntero al primer nodo de la lista.
- $\text{T } e$  – Elemento que vamos a insertar.
- $\text{bool\& enc}$  – Parámetro de salida que indica la acción que tomó la función.
- $\text{int } \text{cmpTT}(\text{T}, \text{T})$  – Función que establece el criterio de ordenamiento.

*Retorna:*  $\text{Node}<\text{T}>^*$  – Dirección del nodo que acabamos de encontrar o insertar.

Ejemplo de uso:

```

bool enc;
Node<int>* p = NULL;

searchAndInsert<int>(p,1,enc,cmpInt); // p->{1}
cout << enc << endl; // Salida: false

searchAndInsert<int>(p,2,enc,cmpInt); // p->{1,2}
cout << enc << endl; // Salida: false

searchAndInsert<int>(p,3,enc,cmpInt); // p->{1,2,3}
cout << enc << endl; // Salida: false

searchAndInsert<int>(p,2,enc,cmpInt); // p->{1,2,3}
cout << enc << endl; // Salida: true

searchAndInsert<int>(p,1,enc,cmpInt); // p->{1,2,3}
cout << enc << endl; // Salida: true

searchAndInsert<int>(p,4,enc,cmpInt); // p->{1,2,3,4}
cout << enc << endl; // Salida: false

```

#### 1.7.1.9. Función sort

*Prototipo:* void sort(Node<T>\*& p, int cmpTT(T,T));

*Descripción:* Ordena la lista direccionada por p según el criterio que establece la función de comparación cmpTT.

*Parámetros:*

- Node<T>\*& p – Puntero al primer nodo de la lista.
- int cmpTT(T,T) – Función que establece el criterio de ordenamiento.

*Retorna:* void.

*Ejemplo de uso:*

```

Node<int>* p = NULL;
add<int>(p,2);
add<int>(p,1);
add<int>(p,3); // p->{2,1,3}

sort<int>(p,cmpInt); // p->{1,2,3}

```

**1.7.1.10. Función isEmpty**

*Prototipo:* `bool isEmpty(Node<T>* p);`

*Descripción:* Indica si la lista direccionada por `p` tiene o no elemento.

*Parámetro:* `Node<T>* p` – Puntero al primer nodo de la lista.

*Retorna:* `bool` – `true` o `false` según la lista tenga o no elementos.

Ejemplo de uso:

```
Node<int>* p = NULL;
add<int>(p,1);
add<int>(p,2);
add<int>(p,3); // p->{1,2,3}

if( !isEmpty<int>(p) )
{
    cout << "la lista tiene elementos" << endl;
}
```

**1.7.1.11. Función free**

*Prototipo:* `void free(Node<T>*& p);`

*Descripción:* Libera la memoria que utiliza lista direccionada por `p`. Asigna `NULL` a `p`.

*Parámetro:* `Node<T>*& p` – Puntero al primer nodo de la lista.

*Retorna:* `void`.

Ejemplo de uso:

```
Node<int>* p = NULL;
add<int>(p,1);
add<int>(p,2);
add<int>(p,3); // p->{1,2,3}

free<int>(p); // p->NULL
```

**1.7.2. API de operaciones sobre pilas (extensión)**

Las siguientes funciones extienden la API de operaciones sobre listas.



**1.7.2.1. Función push**

*Prototipo:* `Node<T>* push (Node<T>* & p, T e) ;`

*Descripción:* Inserta un nodo conteniendo a e al inicio de la lista direccionada por p.

*Parámetros:*

- `Node<T>* & p` – Puntero al primer nodo de la lista.
- `T e` – Elemento que vamos a agregar al inicio de la lista (apilar).

*Retorna:* `Node<T>*`  – Dirección del nodo que contiene al elemento que se agregó.

Ejemplo de uso:

```
Node<int>* p = NULL;
push<int>(p, 1); // p->{1}
push<int>(p, 2); // p->{2,1}
push<int>(p, 3); // p->{3,2,1}
```

**1.7.2.2. Función pop**

*Prototipo:* `T pop (Node<T>* & p) ;`

*Descripción:* Remueve el primer nodo de la lista direccionada por p.

*Parámetro:* `Node<T>* & p` – Puntero al primer nodo de la lista.

*Retorna:* `T` – Elemento que contenía el nodo que fue removido.

Ejemplo de uso:

```
Node<int>* p = NULL;
push<int>(p, 1); // p->{1}
push<int>(p, 2); // p->{2,1}
push<int>(p, 3); // p->{3,2,1}

int e = pop<int>(p); // p->{2,1}
cout << e << endl; // Salida: 3
```

**1.7.3. API de operaciones sobre colas (extensión)**

Las siguientes funciones extienden la API de operaciones sobre listas.

**1.7.3.1. Función enqueue**

**Prototipo:** `Node<T>* enqueue (Node<T>* & p, Node<T>* & q, T e);`

**Descripción:** Agrega el elemento *e* al final la lista direccionada por *q*.

**Parámetros:**

- `Node<T>* & p` – Puntero al primer nodo de la lista.
- `Node<T>* & q` – Puntero al último nodo de la lista.
- `T e` – Elemento que vamos a agregar al final de la lista (*q*).

**Retorna:** `Node<T>*` – Dirección del nodo que contiene al elemento que se agregó.

Ejemplo de uso:

```
Node<int>* p = NULL;
Node<int>* q = NULL;

enqueue<int>(p, q, 1); // p->{1}<-q
enqueue<int>(p, q, 2); // p->{1,2}<-q
enqueue<int>(p, q, 3); // p->{1,2,3}<-q
```

**1.7.3.2. Función enqueue (sobrecarga)**

**Prototipo:** `Node<T>* enqueue (Node<T>* & q, T e);`

**Descripción:** Agrega el elemento *e* al final la lista circular direccionada por *q*.

**Parámetros:**

- `Node<T>* & q` – Puntero al último nodo de la lista circular.
- `T e` – Elemento que vamos a agregar al final de la lista (*q*).

**Retorna:** `Node<T>*` – Dirección del nodo que contiene al elemento que se agregó.

Ejemplo de uso:

```
Node<int>* q = NULL;
enqueue<int>(q, 1); // {1}<-q
enqueue<int>(q, 2); // {1,2}<-q
enqueue<int>(q, 3); // {1,2,3}<-q
```

**1.7.3.3. Función dequeue**

*Prototipo:* T dequeue (Node<T>\* & p, Node<T>\* & q);

*Descripción:* Remueve el primer nodo de la lista direccionada por p.

*Parámetros:*

- Node<T>\* & p - Puntero al primer nodo de la lista.
- Node<T>\* & q - Puntero al último nodo de la lista.

*Retorna:* T - Elemento que contenía el nodo que fue removido.

Ejemplo de uso:

```
Node<int>* p = NULL;
Node<int>* q = NULL;

enqueue<int>(p,q,1); // p->{1}<-q
enqueue<int>(p,q,2); // p->{1,2}<-q
enqueue<int>(p,q,3); // p->{1,2,3}<-q

int e = dequeue<int>(p,q); // p->{2,3}<-q
cout << e << endl; // Salida: 1
```

**1.7.3.4. Función dequeue (sobrecarga)**

*Prototipo:* T dequeue (Node<T>\* & q);

*Descripción:* Remueve el primer nodo de la lista circular direccionada por q.

*Parámetro:* Node<T>\* & q - Puntero al último nodo de la lista circular.

*Retorna:* T - Elemento que contenía el nodo que fue removido.

Ejemplo de uso:

```
Node<int>* q = NULL;
enqueue<int>(q,1); // {1}<-q
enqueue<int>(q,2); // {1,2}<-q
enqueue<int>(q,3); // {1,2,3}<-q

int e = dequeue<int>(q); // {2,3}<-q
cout << e << endl; // Salida: 1
```

### 1.7.4. TAD List

Las siguientes funciones deben programarse en el archivo `List.hpp`.

#### 1.7.4.1. Estructura del TAD

```
template<typename T>
struct List
{
    // implementacion a cargo del estudiante
};
```

Las funciones que describiremos a continuación se deben implementar invocando a los *templates* previamente desarrollados.

```
List<int> lst = list<int>();

// agregamos elementos
listAdd<int>(lst,1);
listAdd<int>(lst,2);
listAdd<int>(lst,3);

// iteramos
listReset<int>(lst);
while( listHasNext<int>(lst) )
{
    int* e = listNext<int>(lst);
    cout << *e << endl;
}

// liberamos
listFree<int>(lst);
```

#### 1.7.4.2. Función list

*Prototipo:* `List<T> list();`

*Descripción:* Función de inicialización.

*Retorna:* `List<T>` – La lista.

**1.7.4.3. Función listAdd**

*Prototipo:*  $T^* \text{ listAdd}(\text{List}<T>\& \text{ lst}, T \text{ e});$

*Descripción:* Agrega un elemento al final de la lista.

*Parámetros:*

- $\text{List}<T>\& \text{ lst}$  - Lista.
- $T \text{ e}$  - Elemento que se agregará al final de la lista.

*Retorna:*  $T^*$  - Dirección de memoria del elemento que se agregó.

**1.7.4.4. Función listAddFirst**

*Prototipo:*  $T^* \text{ listAddFirst}(\text{List}<T>\& \text{ lst}, T \text{ e});$

*Descripción:* Agrega el elemento  $e$  al inicio de la lista.

*Parámetros:*

- $\text{List}<T>\& \text{ lst}$  - Lista.
- $T \text{ e}$  - Elemento que se agregará inicio de la lista.

*Retorna:*  $T^*$  - Dirección de memoria del elemento que se agregó.

**1.7.4.5. Función listRemove**

*Prototipo:*  $T \text{ listRemove}(\text{List}<T>\& \text{ lst}, K \text{ k}, \text{int cmpTK}(T, K));$

*Descripción:* Remueve el elemento que concuerde con  $k$  según la función  $\text{cmpTK}$ .

*Parámetros:*

- $\text{List}<T>\& \text{ lst}$  - Lista.
- $K \text{ k}$  - Elemento que será removido de la lista.

*Retorna:*  $T$  - Elemento que fue removido.

**1.7.4.6. Función listRemoveFirst**

*Prototipo:*  $T \text{ listRemoveFirst}(\text{List}<T>\& \text{ lst});$

*Descripción:* Desenlaza y libera el primer nodo de la lista enlazada, retornando el valor del elemento que contenía.

*Parámetro:*  $\text{List}<T>\& \text{ lst}$  - Lista.

*Retorna:*  $T$  - Elemento que contenía el (ex) primer nodo de la lista.

**1.7.4.7. Función listFind**

*Prototipo:* `T* listFind(List<T> lst, K k, int cmpTK(T, K)) ;`

*Descripción:* Retorna la dirección del primer elemento concordante con `k` según `cmpTK`.

*Parámetros:*

- `List<T> lst` - Lista.
- `K k` - Clave o elemento a buscar.
- `int cmpTK(T, K)` - Función de comparación.

*Retorna:* `T*` - Dirección del elemento encontrado o `NULL` si hubo concordancia.

**1.7.4.8. Función listIsEmpty**

*Prototipo:* `bool listIsEmpty(List<T> lst) ;`

*Descripción:* Indica si la lista está vacía o tiene elementos.

*Parámetro:* `List<T>& lst` - Lista.

*Retorna:* `bool` - `true` si la lista está vacía, `false` si tiene elementos.

**1.7.4.9. Función listSize**

*Prototipo:* `int listSize(List<T> lst) ;`

*Descripción:* Indica cuántos elementos tiene la lista.

*Parámetro:* `List<T>& lst` - Lista.

*Retorna:* `int` - Cantidad de elementos que tiene la lista.

**1.7.4.10. Función listFree**

*Prototipo:* `void listFree(List<T>& lst) ;`

*Descripción:* Libera la memoria que ocupa la lista.

*Parámetro:* `List<T>& lst` - Lista.

*Retorna:* `void`.

**1.7.4.11. Función listOrderedInsert**

*Prototipo:* `T* listOrderedInsert(List<T>& lst  
                                  , T t  
                                  , int cmpTT(T, T)) ;`

*Descripción:* Inserta un elemento según el orden que establece `cmpTT`. La lista debe estar ordenada (según `cmpTT`) o vacía.

*Parámetros:*

- `List<T>& lst` - Lista.
- `T t` - Elemento a insertar.
- `int cmpTT(T, T)` - Función de comparación.

*Retorna:* `T*` - Dirección del elemento insertado.

#### **1.7.4.12. Función `listSort`**

*Prototipo:* `void listSort(List<T> &lst, int cmpTT(T, T));`

*Descripción:* Ordena la lista según el criterio que establece `cmpTT`.

*Parámetros:*

- `List<T>& lst` - Lista.
- `int cmpTT(T, T)` - Función de comparación.

*Retorna:* `void`.

#### **1.7.4.13. Función `listReset`**

*Prototipo:* `void listReset(List<T>& lst);`

*Descripción:* Prepara la lista para iterarla.

*Parámetro:* `List<T>& lst` - Lista.

*Retorna:* `void`.

#### **1.7.4.14. Función `listHasNext`**

*Prototipo:* `bool listHasNext(List<T> lst);`

*Descripción:* Indica si quedan más elementos para seguir iterando la lista.

*Parámetro:* `List<T>& lst` - Lista.

*Retorna:* `bool` - `true` si es posible seguir iterando la lista.

#### **1.7.4.15. Función `listNext`**

*Prototipo:* `T* listNext(List<T>& lst);`

*Descripción:* Retorna la dirección del siguiente elemento de la lista en la iteración.

*Parámetro:* `List<T>& lst` - Lista.

*Retorna:* `T*` - Dirección del siguiente elemento en la iteración.

#### 1.7.4.16. Función `listNext` (sobrecarga)

*Prototipo:* `T* listNext(List<T>& lst, bool& eol);`

*Descripción:* Retorna la dirección del siguiente elemento de la lista en la iteración.

*Parámetros:*

- `List<T>& lst` - Lista.
- `bool& eol` - Indicador de que se llegó al final de la lista (*End Of List*).

*Retorna:* `T*` - Dirección del siguiente elemento en la iteración.

### 1.7.5. TAD Stack

Las siguientes funciones deben programarse en el archivo `Stack.hpp`.

#### 1.7.5.1. Estructura del TAD

```
template<typename T>
struct Stack
{
    // implementacion a cargo del estudiante
};
```

#### 1.7.5.2. Función `stack`

*Prototipo:* `Stack<T> stack();`

*Descripción:* Crea una pila vacía.

*Retorna:* `Stack<T>` - Una pila vacía, lista para usar.

#### 1.7.5.3. Función `stackPush`

*Prototipo:* `T* stackPush(Stack<T>& st, T e);`

*Descripción:* Apila el elemento `e`.

*Parámetros:*



- `Stack<T>& st` - Pila.
- `T e` - Elemento que se apilará.

*Retorna:* `T*` - Dirección de memoria del elemento que se apiló.

#### 1.7.5.4. Función `stackPop`

*Prototipo:* `T stackPop(Stack<T>& st);`

*Descripción:* Desapila un elemento.

*Parámetro:* `Stack<T>& st` - Pila.

*Retorna:* `T` - Elemento que se desapiló.

#### 1.7.5.5. Función `stackIsEmpty`

*Prototipo:* `bool stackIsEmpty(Stack<T> st);`

*Descripción:* Retorna `true` o `false` según la pila tenga elementos o no.

*Parámetro:* `Stack<T> st` - Pila.

*Retorna:* `bool` - `true` o `false` según la pila tenga elementos o no.

#### 1.7.5.6. Función `stackSize`

*Prototipo:* `int stackSize(Stack<T> st);`

*Descripción:* Retorna la cantidad de elementos que tiene la pila.

*Parámetro:* `Stack<T> st` - Pila.

*Retorna:* `int` - Cuántos elementos tiene la pila.

### 1.7.6. TAD Queue

Las siguientes funciones deben programarse en el archivo `Queue.hpp`.

#### 1.7.6.1. Estructura del TAD

```
template<typename T>
struct Queue
{
    // implementacion a cargo del estudiante
};
```

```
};
```

#### 1.7.6.2. Función queue

*Prototipo:* `Queue<T> queue () ;`

*Descripción:* Crea una cola vacía.

*Retorna:* `Queue<T>` – Una cola vacía, lista para usar.

#### 1.7.6.3. Función queueEnqueue

*Prototipo:* `T* queueEnqueue (Queue<T>& q, T e) ;`

*Descripción:* Encola el elemento `e`.

*Parámetros:*

- `Queue<T>& q` - Cola.
- `T e` – Elemento que se encolará.

*Retorna:* `T*` – Dirección de memoria del elemento que se encoló.

#### 1.7.6.4. Función queueDequeue

*Prototipo:* `T queueDequeue (Queue<T>& q) ;`

*Descripción:* Desencola un elemento.

*Parámetro:* `Queue<T>& q` - Cola.

*Retorna:* `T` – Elemento que se desencoló.

#### 1.7.6.5. Función queueIsEmpty

*Prototipo:* `bool queueIsEmpty (Queue<T> q) ;`

*Descripción:* Retorna `true` o `false` según la cola tenga elementos o no.

*Parámetro:* `Queue<T> q` - Cola.

*Retorna:* `bool` - `true` o `false` según la cola tenga elementos o no.

#### 1.7.6.6. Función queueSize

*Prototipo:* `int queueSize (Queue<T> q) ;`

*Descripción:* Retorna la cantidad de elementos que tiene la cola.

*Parámetro:* `Queue<T> q` - Cola.

*Retorna:* `int` – Cuántos elementos tiene la cola.